Scalable Lattice Boltzmann Solvers for CUDA GPU Clusters

Christian Obrecht[a,b,c,*], Frédéric Kuznik[b,c], Bernard Tourancheau[d], Jean-Jacques Roux[b,c]

[a] *EDF R&D, Département EnerBAT, 77818 Moret-sur-Loing Cedex, France*
[b] *Université de Lyon, 69361 Lyon Cedex 07, France*
[c] *INSA-Lyon, CETHIL UMR5008, 69621 Villeurbanne Cedex, France*
[d] *UJF-Grenoble, INRIA, LIG UMR5217, 38041 Grenoble Cedex 9, France*

Abstract

The lattice Boltzmann method (LBM) is an innovative and promising approach in computational fluid dynamics. From an algorithmic standpoint it reduces to a regular data parallel procedure and is therefore well-suited to high performance computations. Numerous works report efficient implementations of the LBM for the GPU, but very few mention multi-GPU versions and even fewer GPU cluster implementations. Yet, to be of practical interest, GPU LBM solvers need to be able to perform large scale simulations. In the present contribution, we describe an efficient LBM implementation for CUDA GPU clusters. Our solver consists of a set of MPI communication routines and a CUDA kernel specifically designed to handle three-dimensional partitioning of the computation domain. Performance measurement were carried out on a small cluster. We show that the results are satisfying, both in terms of data throughput and parallelisation efficiency.

Keywords: GPU clusters, CUDA, lattice Boltzmann method

1. Introduction

A single-GPU based computing device is not proper to solve large scale problems because of the limited amount of on-board memory. However, applications running on multiple GPUs have to face the PCI-E bottleneck, and great care has to be taken in design and implementation to minimise inter-GPU communication. Such constraints may be rather challenging; the well-known MAGMA [15] linear algebra library, for instance, did not support multiple GPUs until version 1.1, two years after the first public release.

The lattice Boltzmann method (LBM) is a novel approach in computational fluid dynamics (CFD), which, unlike most other CFD methods, does not consist

*christian.obrecht@insa-lyon.fr

in directly solving the Navier-Stokes equations by a numerical procedure [9]. Beside many interesting features, such as the ability to easily handle complex geometries, the LBM reduces to a regular data-parallel algorithm and therefore, is well-suited to efficient HPC implementations. As a matter of fact, numerous successful attempts to implement the LBM for the GPU have been reported in the recent years, starting with the seminal work of Li *et al.* in 2003 [10].

CUDA capable computation devices may at present manage up to 6 GB of memory, and the most widespread three-dimensional LBM models require the use of at least 19 floating point numbers per node. Such capacity allows therefore the GPU to process about 8.5×10^7 nodes in single-precision. Taking architectural constraints into account, the former amount is sufficient to store a 416^3 cubic lattice. Although large, such a computational domain is likely to be too coarse to perform direct numerical simulation of a fluid flow in many practical situations as, for instance, urban-scale building aeraulics or thermal modeling of electronic circuit boards.

To our knowledge, the few single-node multi-GPU LBM solvers described in literature all use a one-dimensional (1D) partition of the computation domain, which is relevant in this case since the volume of inter-GPU communication is not likely to be a limiting factor given the small number of involved devices. This option does not require any data reordering, provided the appropriate partitioning direction is chosen, thus keeping the computation kernel fairly simple. For a GPU cluster implementation, on the contrary, a kernel able to run on a three-dimensional (3D) partition seems preferable, since it would both provide more flexibility for load balancing and contribute to reduce the volume of communication.

In the present contribution, we describe an implementation of a lattice Boltzmann solver for CUDA GPU clusters. The core computation kernel is designed so as to import and export data efficiently in each spatial direction, thus enabling the use of 3D partitions. The inter-GPU communication is managed by MPI-based routines. This work constitutes the latest extension to the TheLMA project [1], which aims at providing a comprehensive framework for efficient GPU implementations of the LBM.

The remainder of the paper is structured as follows. In Section 2, we give a description of the algorithmic aspects of the LBM as well as a short review of LBM implementations for the GPU. The third section consists of a detailed description of the implementation principles of the computation kernel and the communication routines of our solver. In the fourth section, we present some performance results on a small cluster. The last section concludes and discusses possible extensions to the present work.

2. State of the art

2.1. Lattice Boltzmann Method

The lattice Boltzmann method is generally carried out on a regular orthogonal mesh with a constant time step δt. Each node of the lattice holds a set of

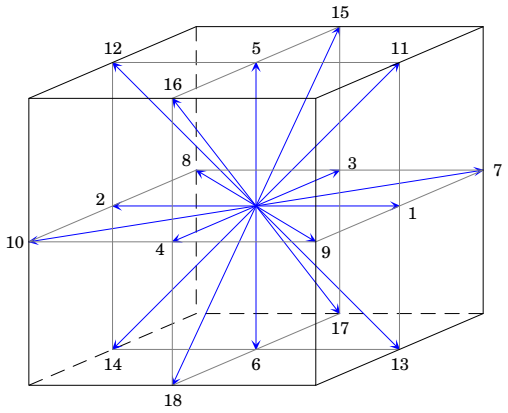


Figure 1: The D3Q19 stencil — *The blue arrows represent the propagation vectors of the stencil linking a given node to some of its nearest neighbours.*

scalars $\{f_\alpha \,|\, \alpha = 0, \ \ldots \ N\}$ representing the local particle density distribution. Each particle density f_α is associated with a particle velocity $\boldsymbol{\xi}_\alpha$ and a *propagation vector* $\boldsymbol{c}_\alpha = \delta t \cdot \boldsymbol{\xi}_\alpha$. Usually the propagation vectors link a given node to one of its nearest neighbours, except for $\boldsymbol{c}_0$ which is null. For the present work, we implemented the D3Q19 propagation stencil illustrated in Fig. 1. This stencil, which contains 19 elements, is the most commonly used in practice for 3D LBM, being the best trade-off between size and isotropy. The governing equation of the LBM at node $\boldsymbol{x}$ and time t writes:

$$\big|f_\alpha(\boldsymbol{x} + \boldsymbol{c}_\alpha, t + \delta t)\big\rangle - \big|f_\alpha(\boldsymbol{x}, t)\big\rangle = \Omega\Big(\big|f_\alpha(\boldsymbol{x}, t)\big\rangle\Big), \tag{1}$$

where $\big|f_\alpha\big\rangle$ denotes the distribution vector and Ω denotes the so-called *collision operator*. The mass density ρ and the momentum $\boldsymbol{j}$ of the fluid are given by:

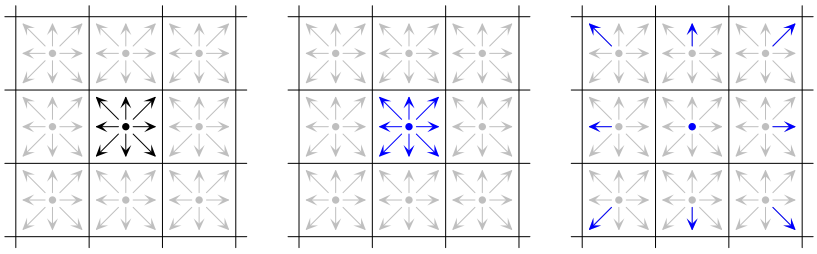
$$\rho = \sum_\alpha f_\alpha \,, \qquad \boldsymbol{j} = \sum_\alpha f_\alpha \boldsymbol{\xi}_\alpha \,. \tag{2}$$

In our solver, we implemented the multiple-relaxation-time collision operator described in [5]. Further information on the physical and numerical aspects of the method are to be found in the aforementioned reference. From an algorithmic perspective, Eq. 1 naturally breaks in two elementary steps:

$$\big|\widetilde{f}_\alpha(\boldsymbol{x}, t)\big\rangle = \big|f_\alpha(\boldsymbol{x}, t)\big\rangle + \Omega\Big(\big|f_\alpha(\boldsymbol{x}, t)\big\rangle\Big), \tag{3}$$

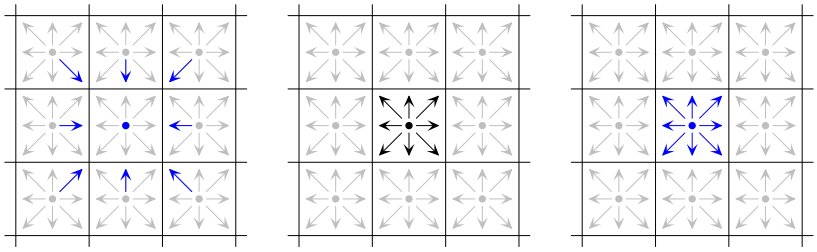
$$\big|f_\alpha(\boldsymbol{x} + \boldsymbol{c}_\alpha, t + \delta t)\big\rangle = \big|\widetilde{f}_\alpha(\boldsymbol{x}, t)\big\rangle. \tag{4}$$

Equation 3 describes the *collision* step in which an updated particle distribution is computed. Equation 4 describes the *propagation* step in which the updated



(a) Initial state (b) Post-collision state (c) Post-propagation state

Figure 2: Collision and propagation — *The collision step is represented by the transition between (a) and (b). The pre-collision particle distribution is drawn in black whereas the post-collision one is drawn in blue. The transition from (b) to (c) illustrates the propagation step in which the updated particle distribution is advected to the neighbouring nodes.*



(a) Initial state (b) Pre-collision state (c) Post-collision state

Figure 3: In-place propagation — *With the in-place propagation scheme, contrary to the out-of-place scheme outlined in Fig. 2, the updated particle distribution of the former time step is advected to the current node before collision.*

particle densities are transferred to the neighbouring nodes. This two-step process is outlined by Fig. 2 (in the two-dimensional case, for the sake of clarity).

2.2. GPU implementations of the LBM

Due to substantial hardware evolution, the pioneering work of Fan *et al.* [6] reporting a GPU cluster LBM implementation is only partially relevant today. The GPU computations were implemented using pre-CUDA techniques that are now obsolete. Yet, the proposed optimisation of the communication pattern still applies, although it was only tested on Gigabyte Ethernet; in future work, we plan to evaluate its impact using InfiniBand interconnect.

In 2008, Tölke and Krafczyk [16] described a single-GPU 3D-LBM implementation using CUDA. The authors mainly try to address the problem induced by misaligned memory accesses. As a matter of fact, with the NVIDIA

G80 GPU available at this time, only aligned and ordered memory transactions could be coalesced. The proposed solution consists in partially performing propagation in shared memory. With the GT200 generation, this approach is less relevant, since misalignment has a lower—though not negligible—impact on performance. As shown in [12], the misalignment overhead is significantly higher for store operations than for read operations. We therefore suggested in [13] to use the in-place propagation scheme outlined by Fig. 3 instead of the ordinary out-of-place propagation scheme illustrated in Fig. 2. From an implementation standpoint, this alternative approach consists in performing the propagation when loading the densities from device memory instead of performing it when storing the densities back. The misalignments therefore only occur on read operations. With the GT200, the gain in performance is about 20%. Moreover, the resulting computation kernel is simpler and leaves the shared memory free for possible extensions.

Further work led us to develop a single-node multi-GPU solver, with 1D partitioning of the computation domain [14]. Each CUDA device is managed by a specific POSIX thread. Inter-GPU communication is carried out using zero-copy transactions to page-locked host memory. Performance and scalability are satisfying with up to 2,482 million lattice node updates per second (MLUPS) and 90.5% parallelisation efficiency on a 384^3 lattice using eight Tesla C1060 computing devices in single-precision.

In their recent paper [17], Wang and Aoki describe an implementation of the LBM for CUDA GPU clusters. The partition of the computation domain may be either one-, two-, or three-dimensional. Although the authors are elusive on this point, no special care seems to be taken to optimise data transfer between device and host memory, and as a matter of fact, performance is quite low. For instance, on a 384^3 lattice with 1D partitioning, the authors report about 862 MLUPS using eight GT200 GPUs in single-precision, i.e. about one third of the performance of our single-node multi-GPU solver using similar hardware. It should also be noted that the given data size for communication per rank, denoted M_{1D}, M_{2D}, and M_{3D}, are at least inaccurate. For the 1D and 2D cases, no account is taken of the fact that for the simple bounce-back boundary condition, no external data is required to process boundary nodes. In the 3D case, the proposed formula is erroneous.

3. Proposed implementation

3.1. Computation kernel

To take advantage of the massive hardware parallelism, our single-GPU and our single-node multi-GPU LBM solvers both assign one thread to each node of the lattice. The kernel execution set-up consists of a two-dimensional grid of one-dimensional blocks, mapping the spatial coordinates. The lattice is stored as a four-dimensional array, the direction of the blocks corresponding to the minor dimension. Two instances of the lattice are kept in device memory, one for even time steps and one for odd time steps, in order to avoid local synchronisation issues. This data layout allows the fetch and store operations to be coalesced since

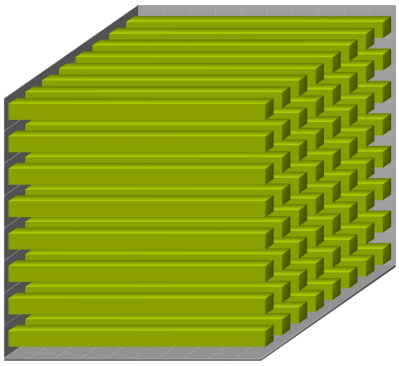


Figure 4: Grid layout for the single-GPU and the single-node multi-GPU LBM solvers — *The execution grid is two-dimensional with one-dimensional blocks spanning the width of the domain (or sub-domain).*

consecutive threads within a warp access consecutive memory locations. It also makes possible, using coalescent zero-copy transactions, to import and export data efficiently at the four sub-domain faces parallel to the blocks, with partial overlapping of communication and computations. For the two sub-domain faces orthogonal to the blocks however, such approach is not practicable since only the first and the last thread within a block would be involved in data exchange, leading to individual zero-copy transactions which, as shown in section 6 of [14], dramatically increase the cost of inter-GPU communication.

A possible solution to extend our computation kernel to support 3D partitions would be to use a specific kernel to handle the interfaces orthogonal to the blocks. Not mentioning the overhead of kernel switching, such an approach does not seem satisfying since the corresponding data is scattered across the array and therefore the kernel would only perform non-coalesced accesses to device memory. As a matter of fact, the minimum data access size is 32 bytes for compute capability up to 1.3, and 128 bytes above, whereas only 4 or 8 bytes would be useful. The cache memory available in devices of compute capability 2.0 and 2.1 is likely to have small impact in this case, taking into account the scattering of the accessed data.

We therefore decided to design a new kernel able to perform propagation and data reordering at once. With this new kernel, blocks are still one-dimensional but, instead of spanning the lattice width, contain only one warp, i.e. $W = 32$ threads (for all existing CUDA capable GPUs). Each block is assigned to a tile of nodes of size $W \times W \times 1$, which imposes for the sub-domain dimensions to be a multiple of W in the x- and y-direction. For a $S_x \times S_y \times S_z$ sub-domain, we therefore use a $(S_x/W) \times (S_y/W) \times S_z$ grid.

The data access pattern is outlined in Fig. 5. For the sake of clarity, let us call *lateral densities* the particle densities crossing the tile sides parallel to the y-direction. Using a D3Q19 stencil, the number of lateral densities is $M = 10$.

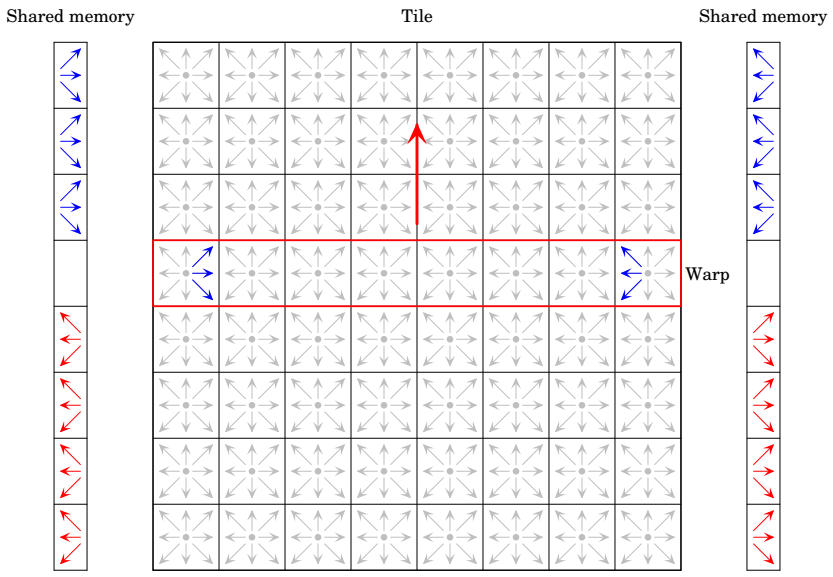


Figure 5: Processing of a tile — *Each tile of nodes is processed row by row by a CUDA block composed of a single warp. Note that we only drew a* 8×8 *tile instead of a* $W \times W$ *tile in order to improve readability. The current row is framed in red and the direction of the processing is indicated by the bold red arrow. The in-coming lateral densities are drawn in blue whereas the out-going ones are drawn in red. These densities are stored in a temporary array hosted in shared memory.*

At each time step, the lateral densities are first loaded into a temporary array in shared memory, then the kernel loops over the tile row by row to process the nodes saving the updated lateral densities in the temporary array, last the updated lateral densities are written back.

In addition to the two instances of the lattice, as for the single-GPU and single-node multi-GPU kernels, the new kernel uses an auxiliary array in device memory to store the lateral densities. As demonstrated by the pseudo-code in Code 1, the data transfer operations issued by the kernel, i.e. the statements on lines 6, 7, 15, 23, and 28, are *coalescent*. These transactions may be either accesses to device memory or, for the nodes located at the interfaces of the sub-domains, zero-copy transactions to communication buffers in host memory. This novel data access pattern makes thus possible to export data efficiently in every spatial direction.

The data transfered by the kernel consist of the densities stored in the lattice instances and the lateral densities stored in the auxiliary array. At each time step, the volume of data read and written amounts to $N \times W^2$ floating point

1. **for each** block B **do**
2. **for each** thread T **do**
3. $\boldsymbol{x}_1 \leftarrow (B_x W, B_y W + T_x, B_z)$
4. $\boldsymbol{x}_2 \leftarrow (B_x W + W - 1, B_y W + T_x, B_z)$
5. **for each** $\alpha \in L$ **do**
6. load $f_\alpha(\boldsymbol{x}_1, t) = \widetilde{f}_\alpha(\boldsymbol{x}_1 - \boldsymbol{c}_\alpha, t - \delta t)$ in shared memory
7. load $f_{\bar{\alpha}}(\boldsymbol{x}_2, t) = \widetilde{f}_{\bar{\alpha}}(\boldsymbol{x}_2 - \boldsymbol{c}_{\bar{\alpha}}, t - \delta t)$ in shared memory
8. **end for**
9. **for** $y = 0$ **to** $W - 1$ **do**
10. $\boldsymbol{x} \leftarrow (B_x W + T_x, B_y W + y, B_z)$
11. **for** $\alpha = 0$ **to** N **do**
12. **if** ($T_x = 0$ and $\alpha \in L$) or ($T_x = W - 1$ and $\bar{\alpha} \in L$) **then**
13. read $f_\alpha(\boldsymbol{x}, t)$ from shared memory
14. **else**
15. load $f_\alpha(\boldsymbol{x}, t) = \widetilde{f}_\alpha(\boldsymbol{x} - \boldsymbol{c}_\alpha, t - \delta t)$
16. **end if**
17. **end for**
18. compute $\left|\widetilde{f}_\alpha(\boldsymbol{x}, t)\right\rangle$
19. **for** $\alpha = 0$ **to** N **do**
20. **if** ($T_x = 0$ and $\bar{\alpha} \in L$) or ($T_x = W - 1$ and $\alpha \in L$) **then**
21. write $\widetilde{f}_\alpha(\boldsymbol{x}, t)$ to shared memory
22. **else**
23. store $\widetilde{f}_\alpha(\boldsymbol{x}, t)$
24. **end if**
25. **end for**
26. **end for**
27. **for each** $\alpha \in L$ **do**
28. store $\widetilde{f}_{\bar{\alpha}}(\boldsymbol{x}_1, t)$ and $\widetilde{f}_\alpha(\boldsymbol{x}_2, t)$
29. **end for**
30. **end for**
31. **end for**

Code 1: Computation kernel — *In this pseudo-code, B_x, B_y, B_z, and T_x denote the indices of block B and thread T; $L = \{1, 7, 9, 11, 13\}$ lists the indices of the propagation vectors with a strictly positive x-component; $\bar{\alpha}$ stands for the direction opposite to α. Note that, for the sake of simplicity, we omitted the processing of boundary conditions.*

numbers[1] per tile for the former, and to $M \times W$ per tile for the later.

The amount of 4-byte (or 8-byte) words read or written per block and per time step is therefore:

$$Q_{\mathrm{T}} = 2\left(19W^2 + 10W\right) = 38W^2 + 20W, \tag{5}$$

and the amount of data read or written in device memory per time step for a $S_x \times S_y \times S_z$ sub-domain is:

$$Q_{\mathrm{S}} = \frac{S_x}{W} \times \frac{S_y}{W} \times S_z \times Q_{\mathrm{T}} = S_x S_y S_z \frac{38W + 20}{W}\,. \tag{6}$$

We therefore see that this approach only increases the volume of device memory accesses by less than 2%, with respect to our former implementations [12], while greatly reducing the number of misaligned transactions.

3.2. Multi-GPU solver

To enable our kernel to run across a GPU cluster, we wrote a set of MPI-based initialisation and communication routines. These routines as well as the new computation kernel were designed as components of the TheLMA framework, which was first developed for our single-node multi-GPU LBM solver. The main purpose of TheLMA is to improve code reusability. It comes with a set of generic modules providing the basic features required by a GPU LBM solver. This approach allowed us to develop our GPU cluster implementation more efficiently.

The execution set-up as well as general parameters such as the Reynolds number of the flow simulation or various option flags, are specified by a configuration file in JSON format [4]. The listing in Code 2 gives an example file for a $2 \times 2 \times 1$ partition running on two nodes. The parameters for each sub-domains, such as the size or the target node and computing device, are given in the `Subdomains` array. The `Faces` and `Edges` arrays specify to which sub-domains a given sub-domain is linked, either through its faces or edges. These two arrays follow the same ordering as the propagation vector set displayed in Fig. 1. Being versatile, the JSON format is well-suited for our application. Moreover, its simplicity makes both parsing and automatic generation straightforward. This generic approach brings flexibility. It allows any LBM solver based on our framework to be tuned to the target architecture.

Our implementation requires the use of an MPI process for each sub-domain to handle. At start, the rank 0 process is responsible for processing the configuration file. Once this file is parsed, the MPI processes register themselves by sending their MPI processor name to the rank 0 process, which in turn assigns an appropriate sub-domain to each of them and sends back all necessary parameters. The processes then perform local initialisation, setting the assigned CUDA device and allocating the communication buffers, which fall into three

[1] A whole segment access is performed for each row and each density, although for some densities the first or the last thread do not issue an individual transaction.

```
{
    "Path": "out",
    "Prefix": "ldc",
    "Re": 1E3,
    "U0": 0.1,
    "Log": true,
    "Duration": 10000,
    "Period": 100,
    "Images": true,
    "Subdomains": [
        {
              "Id": 0,
              "Host": "node00",
              "GPU": 0,
              "Offset": [0, 0, 0],
              "Size": [128, 128, 256],
              "Faces": [   1, null,    2, null, null, null],
              "Edges": [   3, null, null, null, null, null,
                        null, null, null, null, null, null]
        },
        {
              "Id": 1,
              "Host": "node00",
              "GPU": 1,
              "Offset": [128, 0, 0],
              "Size": [128, 128, 256],
              "Faces": [null,    0,    3, null, null, null],
              "Edges": [null,    2, null, null, null, null,
                        null, null, null, null, null, null]
        },
        {
              "Id": 2,
              "Host": "node01",
              "GPU": 0,
              "Offset": [0, 128, 0],
              "Size": [128, 128, 256],
              "Faces": [   3, null, null,    0, null, null],
              "Edges": [null, null,    1, null, null, null,
                        null, null, null, null, null, null]
        },
        {
              "Id": 3,
              "Host": "node01",
              "GPU": 1,
              "Offset": [128, 128, 0],
              "Size": [128, 128, 256],
              "Faces": [null,    2, null,    1, null, null],
              "Edges": [null, null, null,    0, null, null,
                        null, null, null, null, null, null]
        }
    ]
}
```

Code 2: Configuration file

categories: send buffers, receive buffers and read buffers. It is worth noting that both send buffers and read buffers consist of pinned memory allocated using the CUDA API, since they have to be made accessible by the GPU.

The steps of the main computation loop consist of a kernel execution phase and a communication phase. During the first phase, the out-going particle densities are written to the send buffers assigned to the faces *without* performing any propagation as for the densities written in device memory. During the second phase, the following operations are performed:

1. The relevant densities are copied to the send buffers assigned to the edges.
2. Non-blocking send requests are issued for all send buffers.
3. Blocking receive requests are issued for all receive buffers.
4. Once message passing is completed, the particle densities contained in the receive buffers are copied to the read buffers.

This communication phase is outlined in Fig. 6. The purpose of the last operation is to perform propagation for the in-coming particle densities. As a result, the data corresponding to a face and its associated edges is gathered in a single read buffer. This approach avoids misaligned zero-copy transactions, and most important, leads to a simpler kernel since only six buffers at most have to be read. It should be mentioned that the read buffers are allocated using the *write combined* flag to optimise cache usage. According to [7, 8, 11], this setting is likely to improve performance since the memory pages are locked.

4. Performance study

We conducted experiments on an eight-node GPU cluster, each node being equipped with two hexa-core X5650 Intel Xeon CPUs, 36 GB memory, and three NVIDIA Tesla M2070 computing devices; the network interconnect uses QDR InfiniBand. To evaluate raw performance, we simulated a lid-driven cavity [2] in single-precision and recorded execution times for 10,000 time steps using various configurations. Overall performance is good, with at most 8,928 million lattice node updates per second (MLUPS) on a 768^3 lattice using all 24 GPUs. To set a comparison, Wang and Aoki in [17] report at most 7,537 MLUPS for the same problem size using four time as many GPUs. However, it should be mentioned that these results were obtained using hardware of the preceding generation.

The solver was compiled using CUDA 4.0 and OpenMPI 1.4.4. It is also worth mentioning that the computing devices had ECC support enabled. From tests we conducted on a single computing device, we expect the overall performance to be about 20% higher with ECC support disabled.

4.1. Performance model

Our first performance benchmark consisted in running our solver using eight GPUs on a cubic cavity of increasing size. The computation domain is split in a $2 \times 2 \times 2$ regular partition, the size S of the sub-domains ranging from 128 to 288. In addition, we recorded the performance for a single-GPU on a domain of

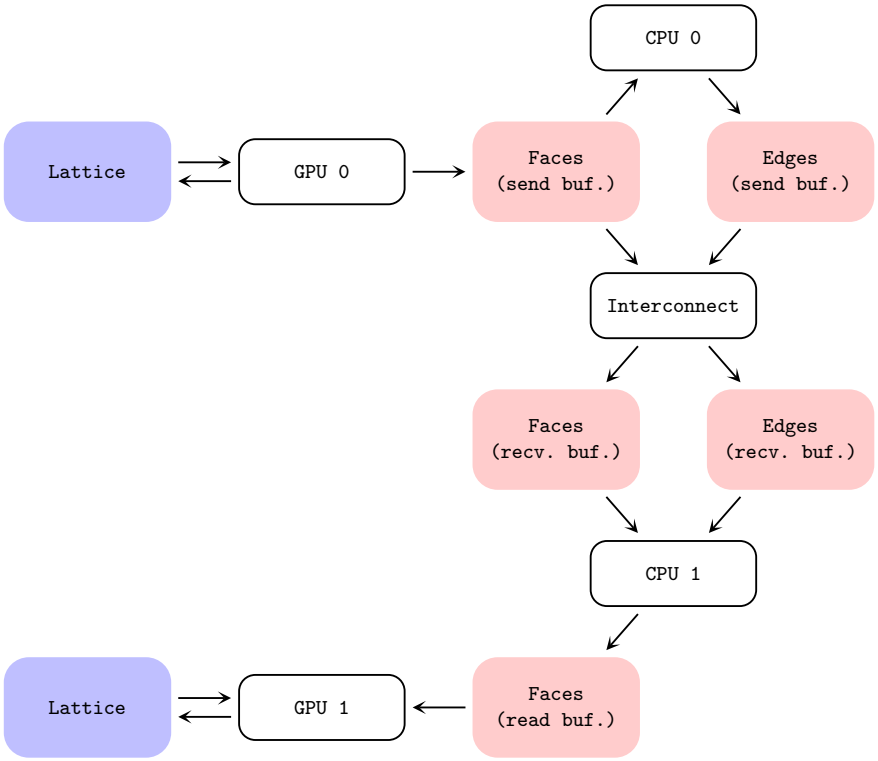


Figure 6: Communication phase — *The upper part of the graph outlines the path followed by data leaving the sub-domain handled by GPU 0. For each face of the sub-domain, the out-going densities are written by the GPU to pinned buffers in host memory. The associated MPI process then copies the relevant densities into the edge buffers and sends both face and edge buffers to the corresponding MPI processes. The lower part of the graph describes the path followed by data entering the sub-domain handled by GPU 1. Once the reception of in-coming densities for faces and edges is completed, the associated MPI process copies the relevant data for each face of the sub-domain into pinned host memory buffers, which are read by the GPU during kernel execution.*

size S, in order to evaluate the communication overhead and the GPU to device memory data throughput. The results are gathered in Tables 1, 2, and 3.

Table 1 shows that the data throughput between GPU and device memory is stable, only slightly increasing with the size of the domain. (Given the data layout in device memory, the increase of the domain size is likely to reduce the amount of L2 cache misses, having therefore a positive impact on data transfer.) We may therefore conclude that the performance of our kernel is communication bound. The last column accounts for the ratio of the data throughput to the maximum sustained throughput, for which we used the value 102.7 GB/s obtained using the `bandwidthTest` program that comes with the CUDA SDK. The obtained ratios are fairly satisfying taking into account the complex data access pattern the kernel must follow.

In Tables 2 and 3, the parallel efficiency and the non-overlapped communication time were computed using the single-GPU results. The efficiency is good with at least 87.3% and appears to benefit from surface-to-volume effects. In Tab. 3, the third column reports the overall data throughput (intra-node and inter-node), the fourth column gives the amount of data transmitted over the interconnect per time step, and the last column reports the corresponding throughput. Both throughputs remain rather stable when the size of the domain increases from 256 to 576, only decreasing by about 20% whereas the communication load increases by a factor of 5. Figure 7 displays the obtained performance results.

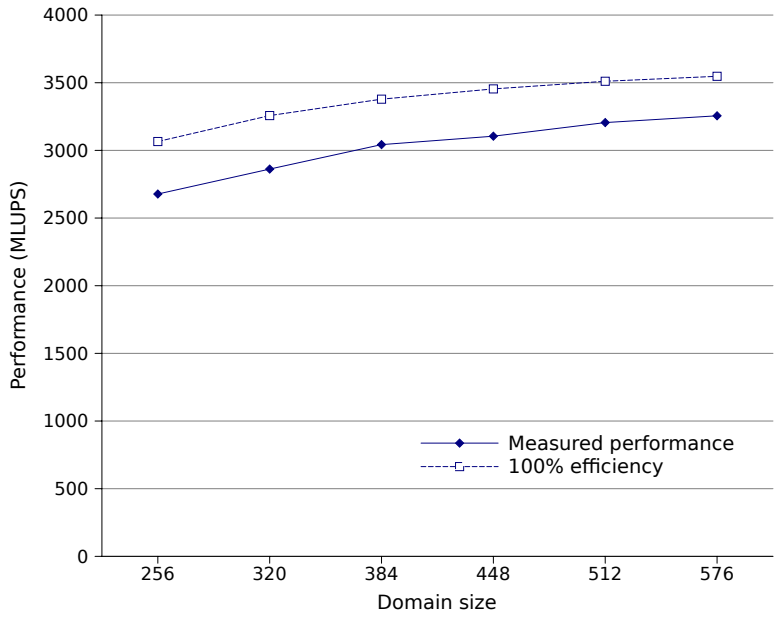


Figure 7: Performance for a $2 \times 2 \times 2$ regular partition

Domain size (S)	Runtime (s)	Performance (MLUPS)	Throughput (GB/s)	Ratio to peak throughput
128	54.7	383.2	59.2	57.6%
160	100.6	407.2	62.9	61.2%
192	167.6	422.3	65.2	63.5%
224	260.3	431.8	66.7	64.9%
256	382.3	438.8	67.8	66.0%
288	538.7	443.4	68.5	66.7%

Table 1: Single-GPU performance

Domain size ($2S$)	Runtime (s)	Perf. (MLUPS)	Parallel efficiency
256	62.7	2,678	87.3%
320	114.5	2,862	87.9%
384	186.9	3,030	89.7%
448	289.6	3,105	89.9%
512	418.7	3,206	91.3%
576	587.0	3,256	91.8%

Table 2: Performance for a $2 \times 2 \times 2$ regular partition

Domain size ($2S$)	Communication (s)	Inter-GPU (GB/s)	Transmission (MB)	Inter-node (GB/s)
256	7.9	9.9	5.0	6.2
320	13.9	8.9	7.8	5.5
384	19.3	9.6	11.3	5.9
448	29.3	8.2	15.3	5.1
512	36.4	8.6	20.0	5.4
576	48.3	8.2	25.3	5.1

Table 3: Data throughput for a $2 \times 2 \times 2$ regular partition

4.2. Scalability

In order to study scalability, both weak and strong, we considered seven different partition types with increasing number of sub-domains. Weak scalability represents the ability to solve larger problems with larger resources whereas strong scalability accounts for the ability to solve a problem faster using more resources. For weak scalability, we used cubic sub-domains of size 128, and for strong scalability, we used a computation domain of constant size 384 with cuboid sub-domains. Table 4 gives all the details of the tested configurations.

For our weak scaling test, we use fixed size sub-domains so that the amount of processed nodes linearly increases with the number of GPUs. We chose a small, although realistic, sub-domain size in order to reduce as much as possible favourable surface-to-volume effects. Since the workload per GPU is fixed, perfect scaling is achieved when the runtime remains constant. The results of the test are gathered in Tab. 5. Efficiency was computed using the runtime of the smallest tested configuration. Figure 8 displays the runtime with respect to the number of GPUs. As illustrated by this diagram, the weak scalability of our solver is satisfying, taking into account that the volume of communication increases by a factor up to 11.5. It is worth noting that in this test, the configuration using 18 GPUs performs better than the configuration using 16 GPUs. It seems that a better node occupancy (i.e. three sub-domains per node instead of two) has a positive impact on performance. However, this hypothesis need to be confirmed by large-scale experiments.

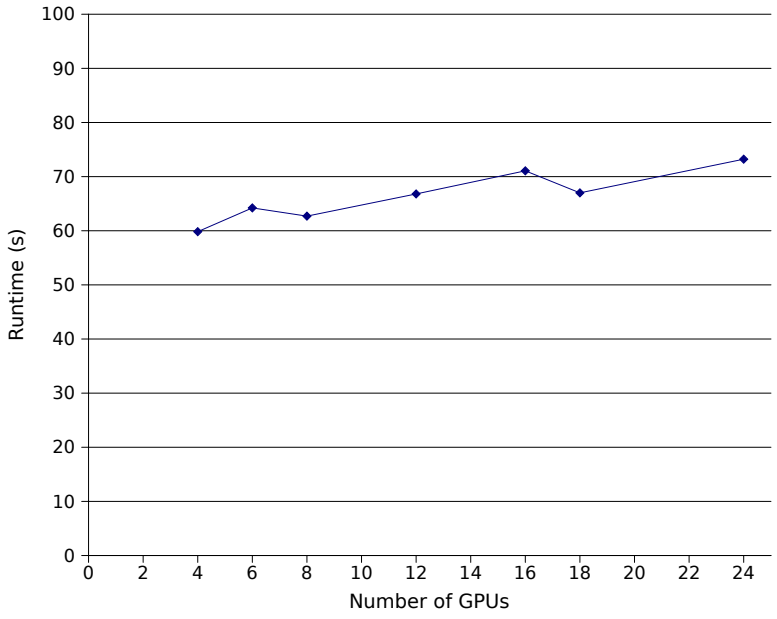


Figure 8: Runtime for the weak scaling test — *Perfect weak scaling would result in an horizontal straight line.*

Number of GPUs	Nodes × GPUs	Partition type	Domain (weak scal.)	Sub-dom. (strong scal.)
4	2×2	$1 \times 2 \times 2$	$128 \times 256 \times 256$	$384 \times 192 \times 192$
6	2×3	$1 \times 3 \times 2$	$128 \times 384 \times 256$	$384 \times 128 \times 192$
8	4×2	$2 \times 2 \times 2$	$256 \times 256 \times 256$	$192 \times 192 \times 192$
12	4×3	$2 \times 3 \times 2$	$256 \times 384 \times 256$	$192 \times 128 \times 192$
16	8×2	$2 \times 4 \times 2$	$256 \times 512 \times 256$	$192 \times 96 \times 192$
18	6×3	$2 \times 3 \times 3$	$256 \times 384 \times 384$	$192 \times 128 \times 128$
24	8×3	$2 \times 4 \times 3$	$256 \times 512 \times 384$	$192 \times 96 \times 128$

Table 4: Configuration details for the scaling tests

Number of GPUs	Runtime (s)	Efficiency	Performance (MLUPS)	Perf. per GPU (MLUPS)
4	59.8	100%	1402	350.5
6	64.2	93%	1959	326.6
8	62.7	95%	2676	334.5
12	66.8	90%	3767	313.9
16	71.1	84%	4721	295.1
18	67.0	89%	5634	313.0
24	73.2	82%	6874	286.4

Table 5: Runtime and efficiency for the weak scaling test

Number of GPUs	Runtime (s)	Efficiency	Performance (MLUPS)	Perf. per GPU (MLUPS)
4	335.0	100%	1690	422.6
6	241.9	92%	2341	390.1
8	186.1	90%	3043	380.3
12	134.7	83%	4204	350.3
16	109.9	76%	5152	322.0
18	98.4	76%	5753	319.6
24	80.3	70%	7053	293.9

Table 6: Runtime and efficiency for the strong scaling test

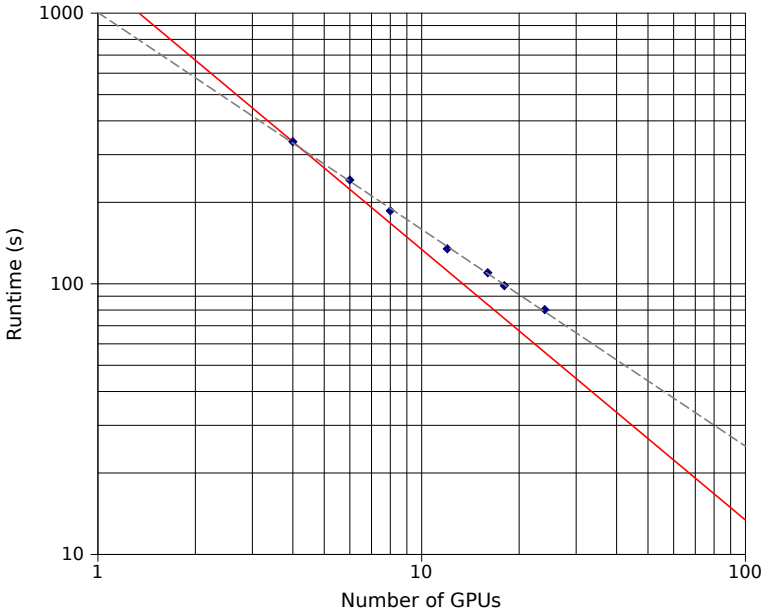


Figure 9: Runtime for the strong scaling test — *Perfect strong scaling is indicated by the solid red line*

In our strong scalability test, we consider a fixed computation domain processed using an increasing number of computing devices. As a consequence the volume of the communication increases by a factor up to three, while the size of the sub-domains decreases, leading to less favourable configurations for the computation kernel. The results of the strong scaling test are given in Tab. 6. The runtime with respect to the number of GPUs is represented in Fig. 9 using a log-log diagram. As shown by the trend-line, the runtime closely obeys a power law, the correlation coefficient for the log-log regression line being below -0.999. The obtained scaling exponent is approximately -0.8, whereas perfect strong scalability corresponds to an exponent of -1. We may conclude that the strong scalability of our code is good, given the fairly small size of the computation domain.

5. Conclusion

In this paper, we describe the implementation of an efficient and scalable LBM solver for GPU clusters. Our code lies upon three main components that were developed for that purpose: a CUDA computation kernel, a set of MPI initialisation routines, and a set of MPI communication routines. The computation kernel's most important feature is the ability to efficiently exchange data in all spatial directions, making possible the use of 3D partitions of the computation

domain. The initialisation routines are designed in order to distribute the workload across the cluster in a flexible way, following the specifications contained in a configuration file. The communication routines manage to pass data between sub-domains efficiently, performing reordering and partial propagation. These new components were devised as key parts of the TheLMA framework[1], whose main purpose is to facilitate the development of LBM solvers for the GPU. The obtained performance on rather affordable hardware such as small GPU clusters makes possible to carry out large scale simulations in reasonable time and at moderate cost. We believe these advances will benefit to many potential applications of the LBM. Moreover, we expect our approach to be sufficiently generic to apply to a wide range of stencil computations, and therefore to be suitable for numerous applications that operate on a regular grid.

Although performance and scalability of our solver is good, we believe there is still room for improvement. Possible enhancements include better overlapping between communication and computation, and more efficient communication between sub-domains. As for now, only transactions to the send and read buffers may overlap kernel computations. The communication phase starts once the computation phase is completed. One possible solution to improve overlapping would be to split the sub-domains in seven zones, six external zones, one for each face of the sub-domains, and one internal zone for the remainder. Processing the external zones first would allow the communication phase to start while the internal zone is still being processed.

Regarding ameliorations to the communication phase, we are considering three paths to explore. First of all, we plan to reinvest the concepts presented in [7] and [8] to improve data transfers involving page-locked buffers. Secondly, we intend to evaluate the optimisation proposed by Fan *et al.* in [6], which consists in performing data exchange in several synchronous steps, one for each face of the sub-domains, the data corresponding to the edges being transfered in two steps. Last, following [3], we plan to implement a benchmark program able to search heuristically efficient execution layouts for a given computation domain and to generate automatically the configuration file corresponding to the most efficient one.

Acknowledgments

The authors wish to thank the INRIA PlaFRIM team for allowing us to test our executables on the Mirage GPU cluster.

References

[1] Thermal LBM on Many-core Architectures. www.thelma-project.info.

[2] S. Albensoeder and H. C. Kuhlmann. Accurate three-dimensional lid-driven cavity flow. *Journal of Computational Physics*, 206(2):536–558, 2005.

[3] R. Clint Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27 (1):3–35, 2001.

[4] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627. Internet Engineering Task Force, 2006.

[5] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society A*, 360:437–451, 2002.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pages 47–58. IEEE, 2004.

[7] P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP: High performance message passing over a cluster of commodity SMPs. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 20–38. ACM, 1999.

[8] P. Geoffray, C. Pham, and B. Tourancheau. A Software Suite for High-Performance Communications on Clusters of SMPs. *Cluster Computing*, 5 (4):353–363, 2002.

[9] X. He and L.-S. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, 1997.

[10] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7):444–456, 2003.

[11] *Compute Unified Device Architecture Programming Guide version 4.0*. NVIDIA, June 2011.

[12] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the LBM on GPUs. In *Lecture Notes in Computer Science 6449, High Performance Computing for Computational Science, VECPAR 2010 Revised Selected Papers*, pages 151–161. Springer, 2011.

[13] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A New Approach to the Lattice Boltzmann Method for Graphics Processing Units. *Computers and Mathematics with Applications*, 12(61):3628–3638, 2011.

[14] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. The TheLMA project: Multi-GPU Implementation of the Lattice Boltzmann Method. *International Journal of High Performance Computing Applications*, 25(3): 295–303, August 2011.

[15] F. Song, S. Tomov, and J. Dongarra. Efficient Support for Matrix Computations on Heterogeneous Multi-core and Multi-GPU Architectures. Technical Report UT-CS-11-668, University of Tennessee, June 2011.

[16] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[17] X. Wang and T. Aoki. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing*, 37(9):521–535, 2011.